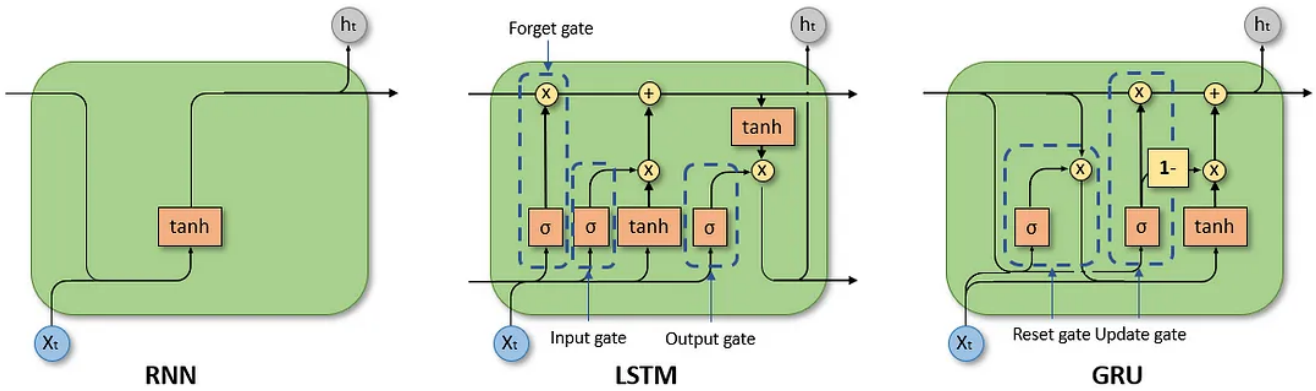
A Brief Introduction to Recurrent Neural Networks

An introduction to RNN, LSTM, and GRU and their implementation

Jonte Dancker · Follow
Published in Towards Data Science
12 min read · Dec 26, 2022



RNN, LSTM, and GRU cells.

If you want to make predictions on sequential or time series data (e.g., text, audio, etc.) traditional neural networks are a bad choice. But why?

In time series data, the current observation depends on previous observations, and thus observations are not independent from each other. Traditional neural networks, however, view each observation as independent as the networks are not able to retain past or historical information. Bascially, they have no memory of what happend in the past.

This led to the rise of Recurrent Neural Networks (RNNs), which introduce the

concept of memory to neural networks by including the dependency between data points. With this, RNNs can be trained to remember concepts based on context, i.e., learn repeated patterns.

But how does an RNN achieve this memory?

RNNs achieve a memory through a feedback loop in the cell. And this is the main difference between a RNN and a traditional neural network. The feed-back loop allows information to be passed within a layer in contrast to feed-forward neural networks in which information is only passed between layers.

RNNs must then define what information is relevant enough to be kept in the memory. For this, different types of RNN evolved:

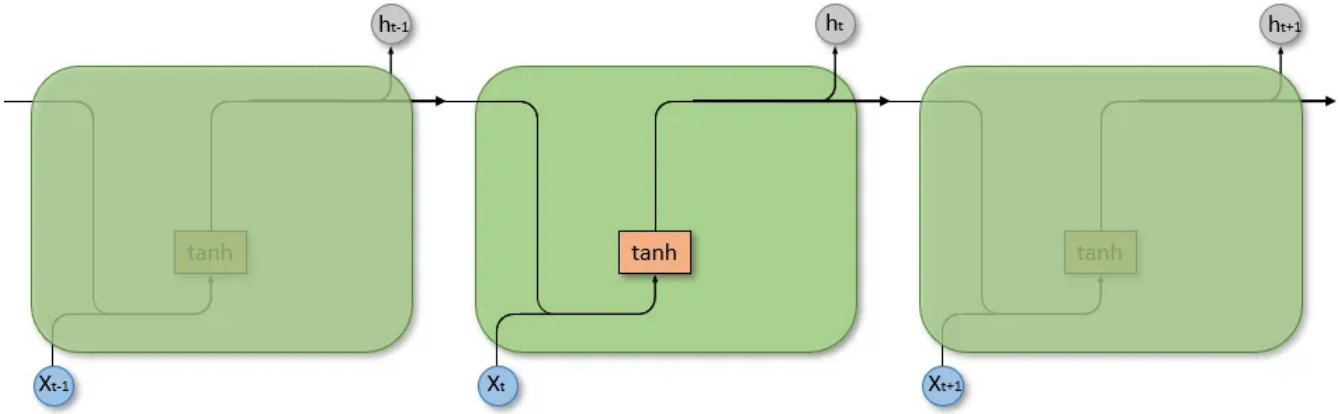
- Traditional Recurrent Neural Network (RNN)
- Long-Short-term-Memory Recurrent Neural Network (LSTM)
- Gated Recurrent Unit Recurrent Neural Network (GRU)

In this article, I give you an introduction to RNN, LSTM, and GRU. I will show you their similarities and differences as well as some advantages and disadvantges. Besides the theoretical foundations I also show you how you can implement each approach in Python using `tensorflow`.

Recurrent Neural Network (RNN)

Through the feedback loop the output of one RNN cell is also used as an input by the same cell. Hence, each cell has two inputs: the past and the present. Using information of the past results in a short term memory.

For a better understanding we unroll/unfold the feedback loop of an RNN cell. The length of the unrolled cell is equal to the number of time steps of the input sequence.



Unfolded Recurrent Neural Network.

We can see how past observations are passed through the unfolded network as a hidden state. In each cell the input of the current time step $\boldsymbol{x}$ (present value), the hidden state $\boldsymbol{h}$ of the previous time step (past value) and a bias are combined and then limited by an activation function to determine the hidden state of the current time step.

$$\boldsymbol{h}_t = \tanh\left(\boldsymbol{W}_x\, \boldsymbol{x}_t + \boldsymbol{W}_h\, \boldsymbol{h}_{t-1} + \boldsymbol{b}\right)$$

Here, the small, bold letters represent vectors while the captial, bold letters represent matrices.

The weights $\boldsymbol{W}$ of the RNN are updated through a backpropagation in time (BPTT) algorithm.

RNNs can be used for one-to-one, one-to-many, many-to-one, and many-to-many predictions.

Advantages of RNNs

Due to their shortterm memory RNNs can handle sequential data and identify patterns in the historical data. Moreover, RNNs are able to handle inputs of varying length.

Disadvantages of RNNs

The RNN suffers from the vanishing gradient descent. In this, the gradients that are

used to update the weights during backpropagation become very small. Multiplying weights with a gradient that is close to zero prevents the network from learning new weights. This stopping of learning results in the RNN forgetting what is seen in longer sequences. The problem of vanishing gradient descent increases the more layers the network has.

As the RNN only keeps recent information, the model has problems to consider observations which lie far in the past. The RNN, thus, tends to loose information over long sequences as it only stores the latest information. Hence, the RNN has only a short-term but not a long-term memory.

Moreover, as the RNN uses backpropagation in time to update weights, the network also suffers from exploding gradients and, if ReLu activation functions are used, from dead ReLu units. The first might lead to convergence issues while the latter might stop the learning.

Implementation of RNNs in tensorflow

We can easily implement a RNN in Python using `tensorflow`. For this, we use the `Sequential` model which allows us to stack layers of RNN, i.e., the `SimpleRNN` layer class, and the `Dense` layer class.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.optimizers import Adam
```

Importing the optimizer is not necessary as long as we want to use the default parameters. However, if we want to customize any parameters of the optimizer we need to import the optimizer as well.

To build the network, we define a `Sequential` model and then use the `add()` method to add the RNN layers. To add a RNN layer, we use the `SimpleRNN` class and pass parameters, such as the number of units, the dropout rate or the activation function. For our first layer we can also pass the shape of our input sequence.

If we stack RNN layers, we need to set the `return_sequence` parameter of the previous layer to `True`. This ensures that the output of the layer has the right format for the next RNN layer.

To generate an output we use a `Dense` layer as our last layer, passing the number of outputs.

```
# define parameters
n_timesteps, n_features, n_outputs = X_train.shape[1], X_train.shape[2], y_trai

# define model
rnn_model = Sequential()
rnn_model.add(SimpleRNN(130, dropout=0.2, return_sequences=True, input_shape=(n
rnn_model.add(SimpleRNN(110, dropout=0.2, activation="tanh", return_sequences=T
rnn_model.add(SimpleRNN(130, dropout=0.2, activation="tanh", return_sequences=T
rnn_model.add(SimpleRNN(100, dropout=0.2, activation="sigmoid", return_sequence
rnn_model.add(SimpleRNN(40, dropout=0.3, activation="tanh"))
rnn_model.add(Dense(n_outputs))
```

After we have defined our RNN, we can compile the model using the `compile()` method. Here, we pass the loss function and the optimizer we want to use. `tensorflow` provides some built-in loss functions and optimizers.

```
rnn_model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=0.001
```

Before we train the RNN, we can have a look at the model and the number of parameters, using the `summary()` method. This can give us and overview about the complexity of our model.

We train the model using the `fit()` method. Here, we need to pass the training data and different parameters to customize the training, including the number of epochs, the batch size, a validation split, and an early stopping.

```
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
rnn_model.fit(X_train, y_train, epochs=30, batch_size=32, validation_split=0.2,
```

To make predictions on our test data set or on any unseen data, we can use the `predict()` method. The `verbose` parameter just states if we want to get any information on the status of the prediction process. In this case, I did not want any print out of the status.

```
y_pred = rnn_model.predict(X_test, verbose=0)
```

Hyperparameter tuning for RNNs in tensorflow

As we can see the implementation of an RNN is pretty straightforward. Finding the right hyperparameters, such as number of units per layer, dropout rate or activation function, however, is much harder.

But instead of varying the hyperparameter manually, we can use the `keras-tuner` library. The library has four tuners, `RandomSearch`, `Hyperband`, `BayesianOptimization`, and `Sklearn`, to identify the right hyperparameter combination from a given search space.

To run the tuner we first need to import `tensorflow` and the Keras Tuner.

```
import tensorflow as tf
import keras_tuner as kt
```

We then build the model for hypertuning, in which we define the hyperparameter search space. We can build the hypermodel using a function, in which we build the model in the same way as above described. The only difference is that we add the

search space for each hyperparameter we want to tune. In the example below, I want to tune the number of units, the activation function, and the dropout rate for each RNN layer.

```
def build_RNN_model(hp):

    # define parameters
    n_timesteps, n_features, n_outputs = X_train.shape[1], X_train.shape[2], y_

    # define model
    model = Sequential()

    model.add(SimpleRNN(hp.Int('input_unit',min_value=50,max_value=150,step=20)
    model.add(SimpleRNN(hp.Int('layer 1',min_value=50,max_value=150,step=20), a
    model.add(SimpleRNN(hp.Int('layer 2',min_value=50,max_value=150,step=20), a
    model.add(SimpleRNN(hp.Int('layer 3',min_value=20,max_value=150,step=20), a
    model.add(SimpleRNN(hp.Int('layer 4',min_value=20,max_value=150,step=20), a

    # output layer
    model.add(Dense(n_outputs))

    model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=1e-3)

    return model
```

To define the search space for each variable we can use different methods, such as `hp.Int`, `hp.Float`, and `hp.Choice`. The first two are very similar their use. We give them a name, a minimum value, a maximum value, and a step size. The name is used to identify the hyperparameter while the minimum and maximum value define our range of values. The step parameters defines the values in the range we use for the tuning. The `hp.Choice` can be used to tune categorical hyperparameters such as the activation function. Here, we only have to pass a list of the choices we want to test.

After we have built our hypermodel, we need to instantiate the tuner and perform the hypertuning. Although we can choose between different algorithms for the tuning, their instantiation is very similar. We generally need to specify the objective to optimize and the maximum number of epochs to train. Here, it is recommended

to set the epochs to a number which is slightly higher than our expected number of epochs and then use early stopping.

For example, if we want to use the `Hyperband` tuner and the validation loss as the objective we can build the tuner as

```
tuner = kt.Hyperband(build_RNN_model,
                     objective="val_loss",
                     max_epochs=100,
                     factor=3,
                     hyperband_iterations=5,
                     directory='kt_dir',
                     project_name='rnn',
                     overwrite=True)
```

Here, I also passed the directory in which the results shall be stored and how often the tuner shall iterate over the full Hyperband algorithm.

After we have instantiated the tuner, we can use the `search()` method to perform the hyperparameter tuning.

```
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
tuner.search(X_train, y_train, validation_split=0.2, callbacks=[stop_early])
```

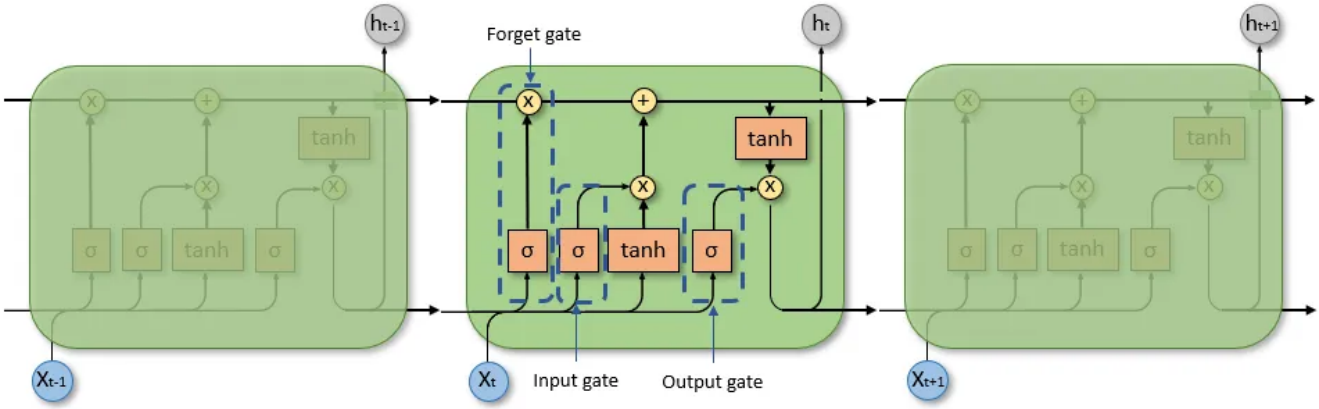
To extract the optimal hyperparameters, we can then use the `get_best_hyperparameters()` method and use the `get()` method and the name of each hyperparameter we tuned.

```
best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]
print(f"input: {best_hps.get('input_unit')}")
print(f"input dropout: {best_hps.get('in_dropout')}")
```

Long-Short-Term-Memory (LSTM)

LSTMs are a special type of RNNs which tackle the main problem of simple RNNs, the problem of vanishing gradients, i.e., the loss of information that lies further in

the past.



Unfolded Long-Short-Term-Memory cell.

The key to LSTMs is the **cell state**, which is passed from the input to the output of a cell. Thus, the cell state allows information to flow along the entire chain with only minor linear actions through three gates. Hence, the cell state represents the long-term memory of the LSTM. The three gates are called the forget gate, input gate, and ouput gate. These gates work as filters and control the flow of information and determine which information is kept or disregarded.

The **forget gate** decides how much of the long-term memory shall be kept. For this, a sigmoid function is used which states the importance of the cell state. The output varies between 0 and 1 and states how much information is kept, i.e., 0, keep no information and 1, keep all information of the cell state. The output is determined by combining the current input $\boldsymbol{x}$, the hidden state $\boldsymbol{h}$ of the previous time step, and a bias $\boldsymbol{b}$.

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_{f,x}\,\boldsymbol{x}_t + \boldsymbol{W}_{f,h}\,\boldsymbol{h}_{t-1} + \boldsymbol{b}_f\right)$$

The **input gate** decides which information shall be added to the cell state and thus the long-term memory. Here, a sigmoid layer decides which values are updated.

$$\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_{i,x}\,\boldsymbol{x}_t + \boldsymbol{W}_{i,h}\,\boldsymbol{h}_{t-1} + \boldsymbol{b}_i\right)$$

The **output gate** decides which parts of the cell state build the output. Hence, the output gate is responsbile for the short-term memory.

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_{o,x}\ \boldsymbol{x}_t + \boldsymbol{W}_{o,h}\ \boldsymbol{h}_{t-1} + \boldsymbol{b}_o\right)$$

As can be seen, all three gates are represented by the same function. Only the weights and biases differ. The cell state is updated through the forget gate and the input gate.

$$\boldsymbol{c}_t = \left(\boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \tanh\left(\boldsymbol{h}_{t-1}\right)\right)$$

The first term in the above equation determines how much of the long-term memory is kept while the second terms adds new information to the cell state.

The hidden state of the current time step is then determined by the output gate and a tanh function which limits the cell state between -1 and 1.

$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh\left(\boldsymbol{c}_t\right)$$

Advantages of LSTMs

The advantages of the LSTM are similar to RNNs with the main benefit being that they can capture patterns in the long-term and short-term of a sequence. Hence, they are the most used RNNs.

Disadvantages of LSTMs

Due to their more complex structure, LSTMs are computationally more expensive, leading to longer training times.

As the LSTM also uses the backpropagation in time algorithm to update the weights, the LSTM suffers from the disadvantages of the backpropagation (e.g., dead ReLu elements, exploding gradients).

Implementation of LSTMs in tensorflow

The implementation of LSTMs in `tensorflow` is very similar to a simple RNN. The

only difference is that we import the `LSTM` class instead of the `SimpleRNN` class.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.optimizers import Adam
```

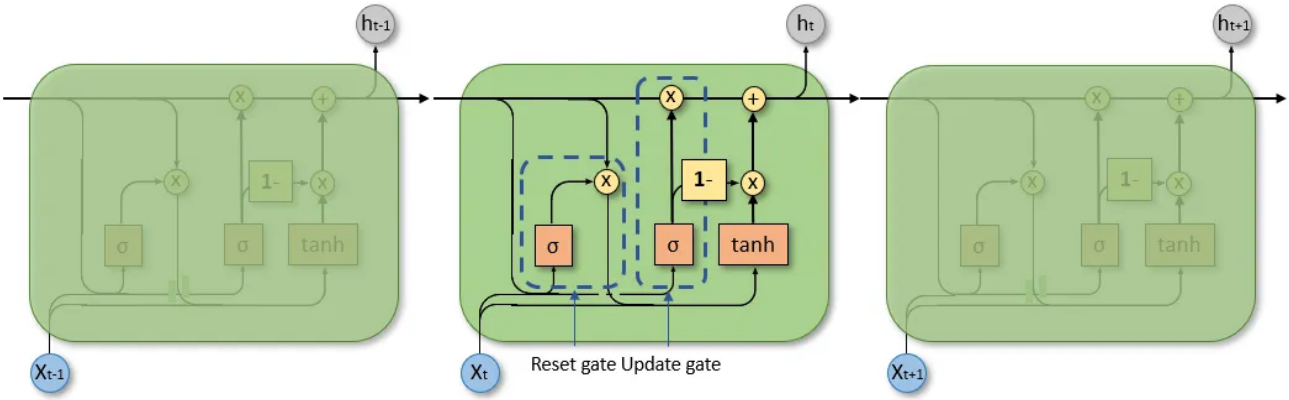
We can the put together the LSTM network in the same way as the simple RNN.

```
# define parameters
n_timesteps, n_features, n_outputs = X_train.shape[1], X_train.shape[2], y_trai

# define model
lstm_model = Sequential()
lstm_model.add(LSTM(130, return_sequences=True, dropout=0.2, input_shape=(n_tim
lstm_model.add(LSTM(70, activation="relu", dropout=0.1, return_sequences=True))
lstm_model.add(LSTM(100, activation="tanh", dropout=0))

# output layer
lstm_model.add(Dense(n_outputs, activation="tanh"))

lstm_model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=0.00

stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
lstm_model.fit(X_train, y_train, epochs=30, batch_size=32, validation_split=0.2
```

The hyperparameter tuning is also the same as for the simple RNN. Hence, we only need to make minor changes to the code snippets I have shown above.

Gated Recurrent Unit (GRU)

Similar to LSTMs, the GRU solves the vanishing gradient problem of simple RNNs. The difference to LSTMs, however, is that GRUs use fewer gates and do not have a separate internal memory, i.e., cell state. Hence, the GRU solely relies on the hidden state as a memory, leading to a simpler architecture.



Unfolded Gated Recurrent Unit (GRU).

The **reset gate** is responsible for the short-term memory as it decides how much past information is kept and disregarded.

$$\boldsymbol{r}_t = \sigma\left(\boldsymbol{W}_{r,x}\ \boldsymbol{x}_t + \boldsymbol{W}_{r,h}\ \boldsymbol{h}_{t-1} + \boldsymbol{b}_r\right)$$

The values in the vector $\boldsymbol{r}$ are bounded between 0 and 1 by a sigmoid function and depend on the hidden state $\boldsymbol{h}$ of the previous time step and the current input $\boldsymbol{x}$. Both are weighted using the weight matrices $\boldsymbol{W}$. Furthermore, a bias $\boldsymbol{b}$ is added.

The **update gate**, in contrast, is responsible for the long-term memory and is comparable to the LSTM's forget gate.

$$\boldsymbol{u}_t = \sigma\left(\boldsymbol{W}_{u,x}\ \boldsymbol{x}_t + \boldsymbol{W}_{u,h}\ \boldsymbol{h}_{t-1} + \boldsymbol{b}_u\right)$$

As we can see the only difference between the reset and update gate are the weights $\boldsymbol{W}$.

The hidden state of the current time step is determined based on a two step process. First, a candidate hidden state is determined. The candidate state is a combination of the current input and the hidden state of the previous time step and an activation function. In this example, a tanh function is used. The influence of the previous hidden state on the candidate hidden state is controlled by the reset gate

$$\hat{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_{g,x}\ \boldsymbol{x}_t + \boldsymbol{W}_{g,h}\left(\boldsymbol{r}_t \circ \boldsymbol{h}_{t-1}\right) + \boldsymbol{b}_g\right)$$

In the second step, the candidate hidden state is combined with the hidden state of the previous time step to generate the current hidden state. How the previous hidden state and the candidate hidden state are combined is determined by the update gate.

$$\boldsymbol{h}_t = \boldsymbol{u}_t \circ \boldsymbol{h}_{t-1} + \left(1 - \boldsymbol{u}_t\right) \circ \hat{\boldsymbol{h}}_t$$

If the update gate gives a value of 0 then the previous hidden state is completly disregarded and the current hidden state is equal to the candidate hidden state. If the update gate gives a value of one, it is vice versa.

Advantages of GRUs

Due to the simpler architecture compared to LSTMs (i.e., two instead of three gates and one state instead of two), GRUs are computationally more efficent and faster to train as they need less memory.

Moreover, GRUs haven proven to be more efficient for smaller sequences.

Disadvantages of GRUs

As GRUs do not have a separate hidden and cell state they might not be able to consider observations as far into the past as the LSTM.

SImilar to the RNN and LSTM, the GRU also might suffer from the disadvantages of the backpropagation in time to update the weights, i.e., dead ReLu elements, exploding gradients.

Implementation of GRUs in tensorflow

As for the LSTM, the implementation of GRU is very similar to simple RNN. We only need to import the `GRU` class while the rest stays the same.

```
from tensorflow.keras import Sequential
```

```
from tensorflow.keras.layers import GRU, Dense
from tensorflow.keras.optimizers import Adam

# define parameters
n_timesteps, n_features, n_outputs = X_train.shape[1], X_train.shape[2], y_trai

# define model
gru_model = Sequential()
gru_model.add(GRU(90,return_sequences=True, dropout=0.2, input_shape=(n_timeste
gru_model.add(GRU(150, activation="tanh", dropout=0.2, return_sequences=True))
gru_model.add(GRU(60, activation="relu", dropout=0.5))
gru_model.add(Dense(n_outputs))

gru_model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=0.001

stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
gru_model.fit(X_train, y_train, epochs=30, batch_size=32, validation_split=0.2,
```

Conclusion

Recurrent Neural Networks introduce a memory into neural networks. With this, the dependency of observations in sequential and time series data is included in our prediction. The same applies to the hyperparameter tuning.

In this article, I have shown you three types of Recurrent Neural Networks, i.e., simple RNN, LSTM and GRU. I have shown you how they work, what their advantages and disadvantages are, and how you can implement them in Python using `tensorflow`.

Please let me know what you think about the article!

*All images unless otherwise noted are by the author.

Recurrent Neural Network | Machine Learning | Time Series Forecasting | Lstm | TensorFlow

Follow

Written by Jonte Dancker

151 Followers · Writer for Towards Data Science

Data Science Enthusiast | Writing about my data science side projects and sharing my learnings

More from Jonte Dancker and Towards Data Science

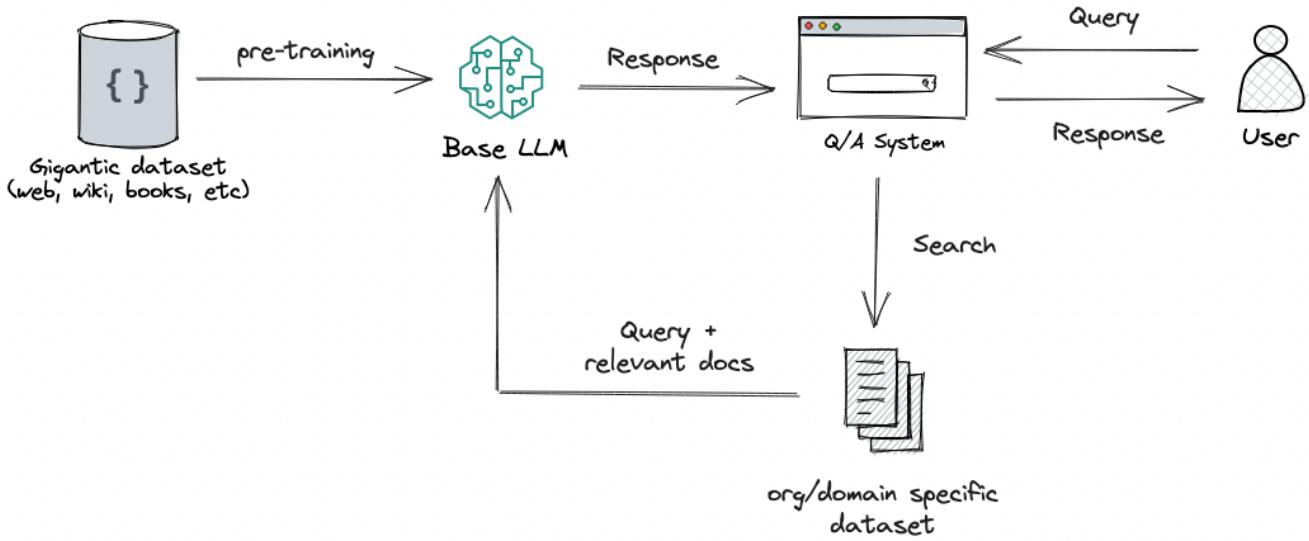
Jonte Dancker in Towards Data Science

A Brief Introduction to SciKit Pipelines

And why you should start using them.

7 min read · Aug 24

32 1



Heiko Hotz in Towards Data Science

RAG vs Finetuning — Which Is the Best Tool to Boost Your LLM Application?

The definitive guide for choosing the right method for your use case

· 19 min read · Aug 24

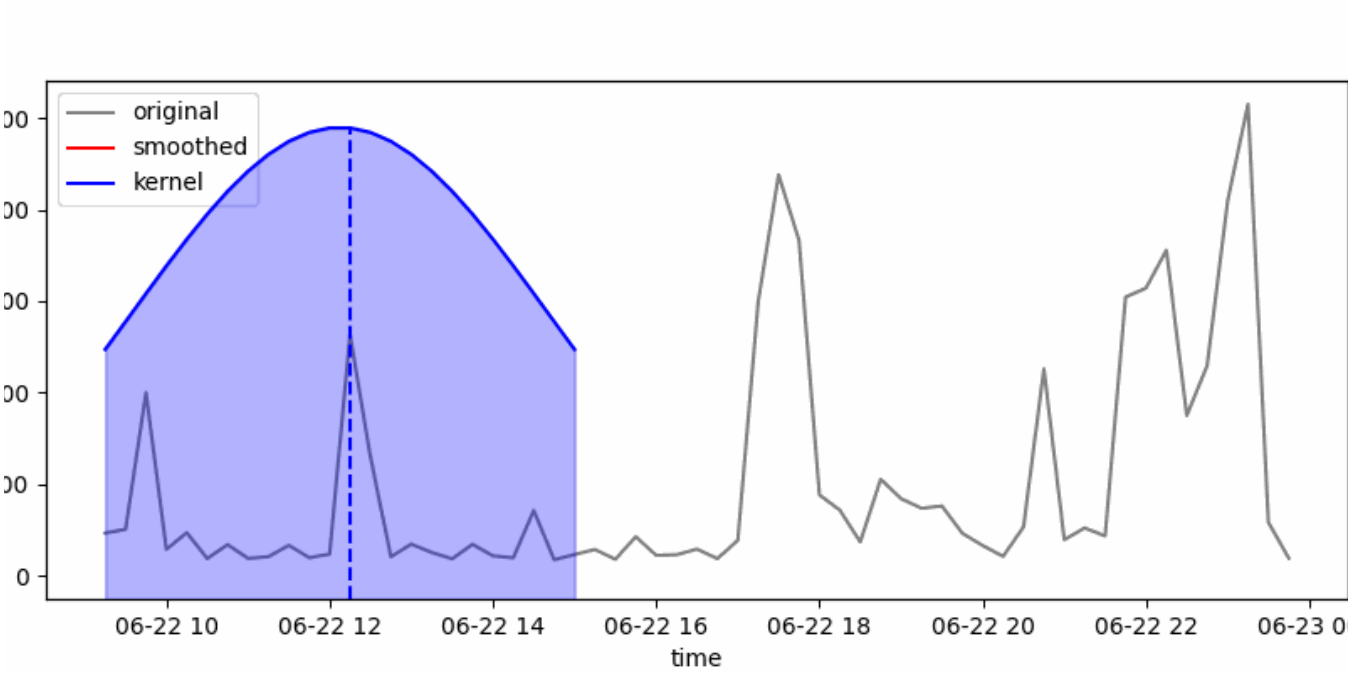
2K 16

Giuseppe Scalamogna in Towards Data Science

New ChatGPT Prompt Engineering Technique: Program Simulation

A potentially novel technique for turning a ChatGPT prompt into a mini-app.

9 min read · Sep 3

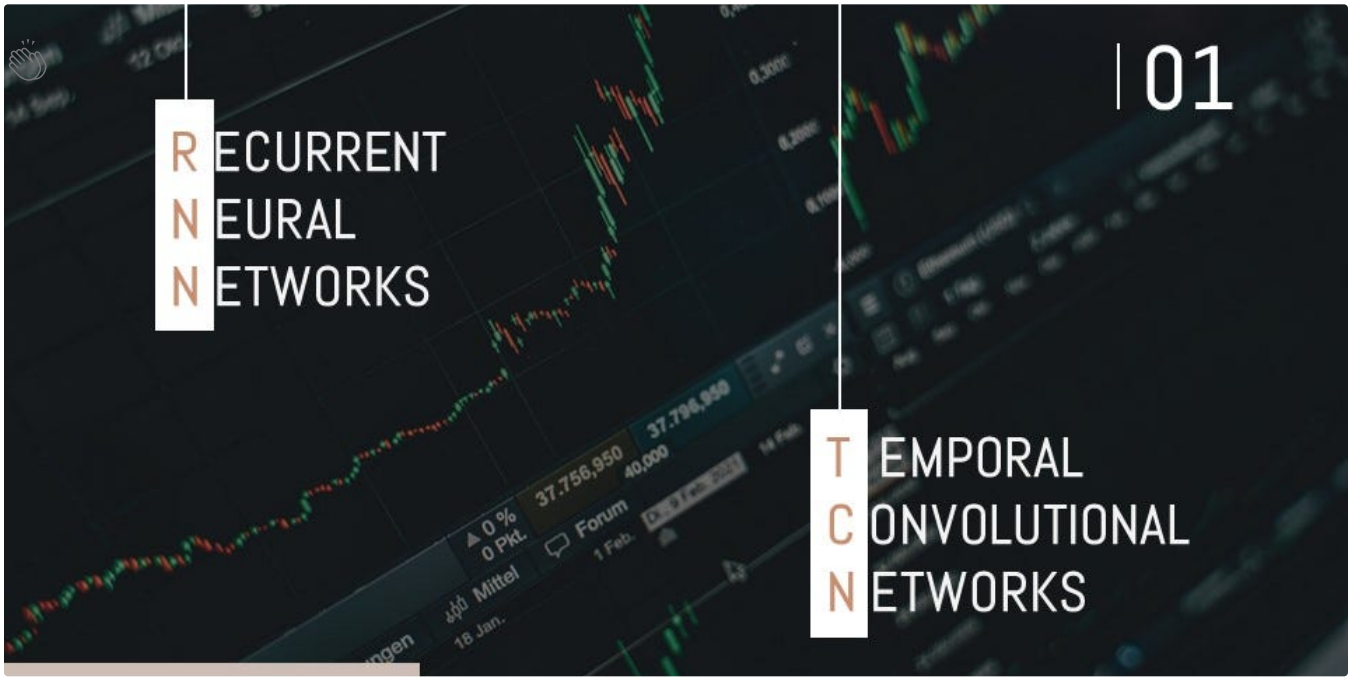


Jonte Dancker

A brief introduction to time series smoothing

You do not know what time series smoothing is or you want to know how to implement time series smoothing in Python?

9 min read · Sep 27, 2022



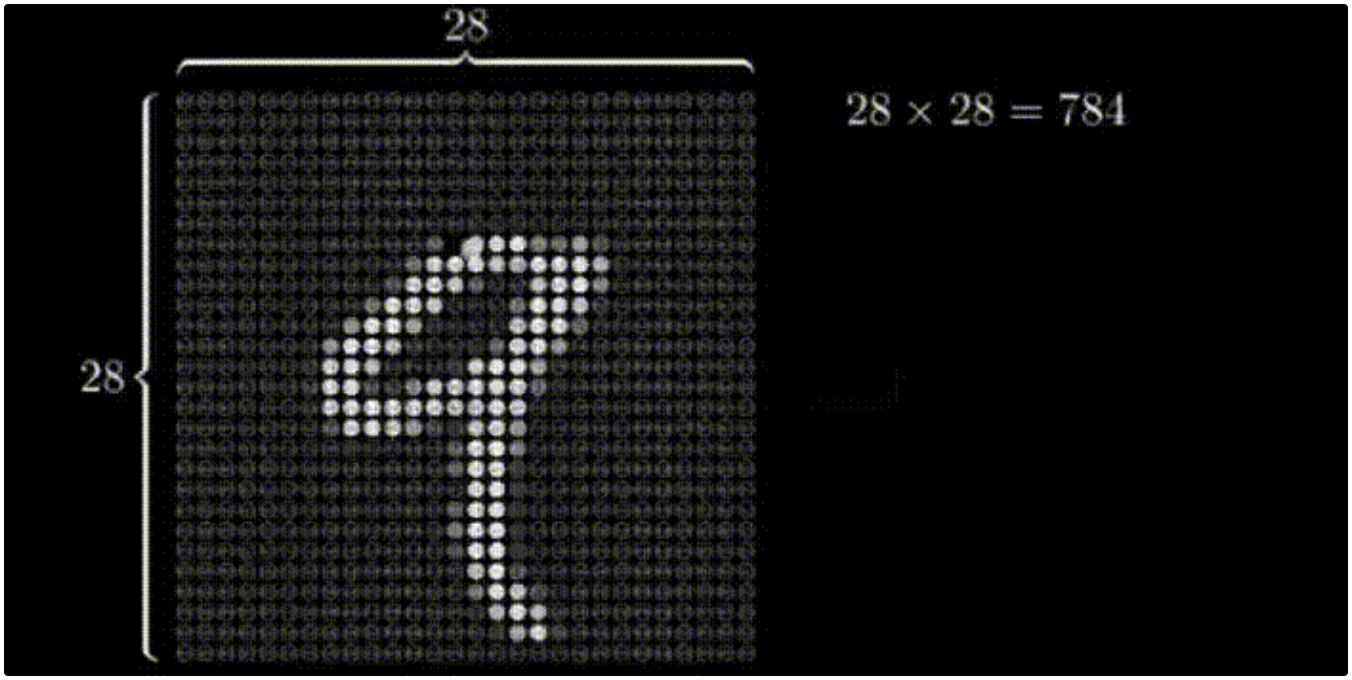
Ricardo Colindres

Temporal Convolutional and Recurrent Neural Networks for Sequence Modeling

Hello! I made this presentation on Temporal Convolutional and Recurrent Network and I would like to share it for those of you just getting...

13 min read · Aug 2

1



Sadaf Saleem

Neural Networks in 10mins. Simply Explained!

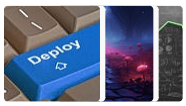
What are Neural Networks?

9 min read · May 15

113 1

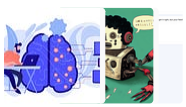
Lists

Practical Guides to Machine Learning
10 stories · 469 saves

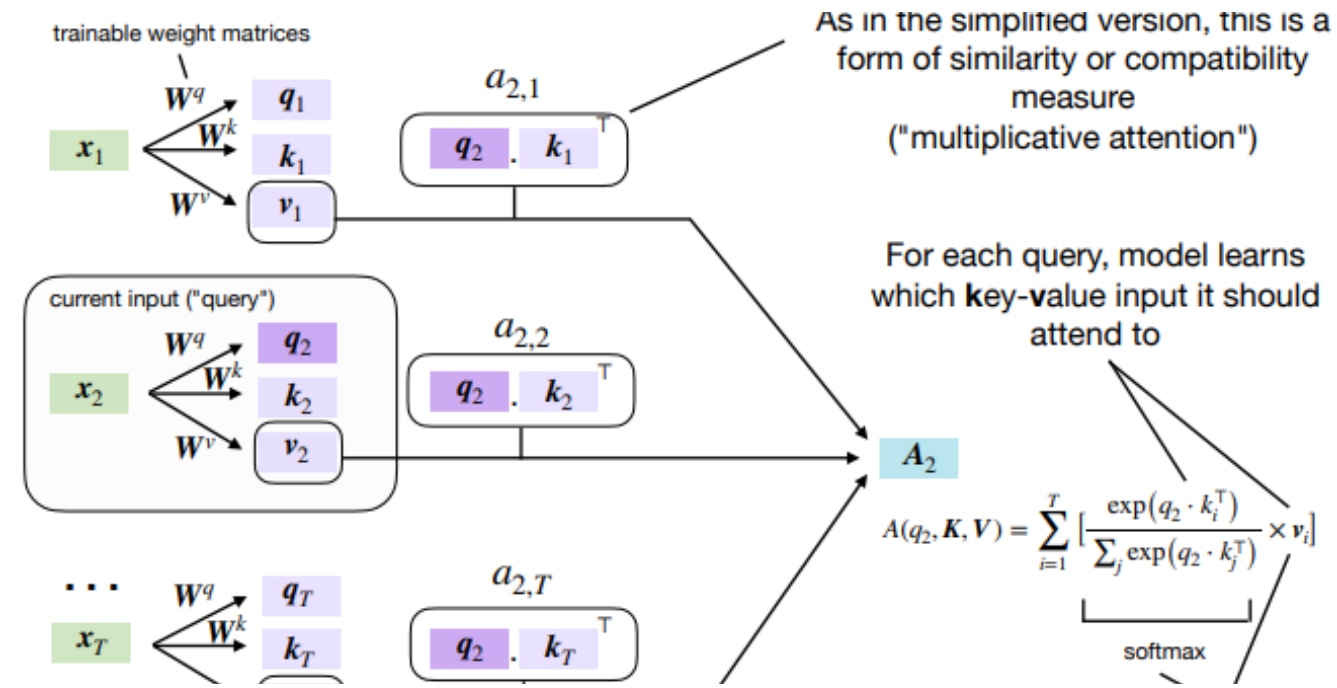


Predictive Modeling w/ Python
20 stories · 405 saves

Natural Language Processing
628 stories · 237 saves



The New Chatbots: ChatGPT, Bard, and Beyond
13 stories · 123 saves



Zain ul Abideen

Attention Is All You Need: The Core Idea of the Transformer

An overview of the Transformer model and its key components.

6 min read · Jun 26

59

Salvatore Raieli in Level Up Coding

Tabula Rasa: Why Do Tree-Based Algorithms Outperform Neural Networks

Tree-based algorithms are the winner in tabular data: Why?

· 19 min read · Sep 14

903 11

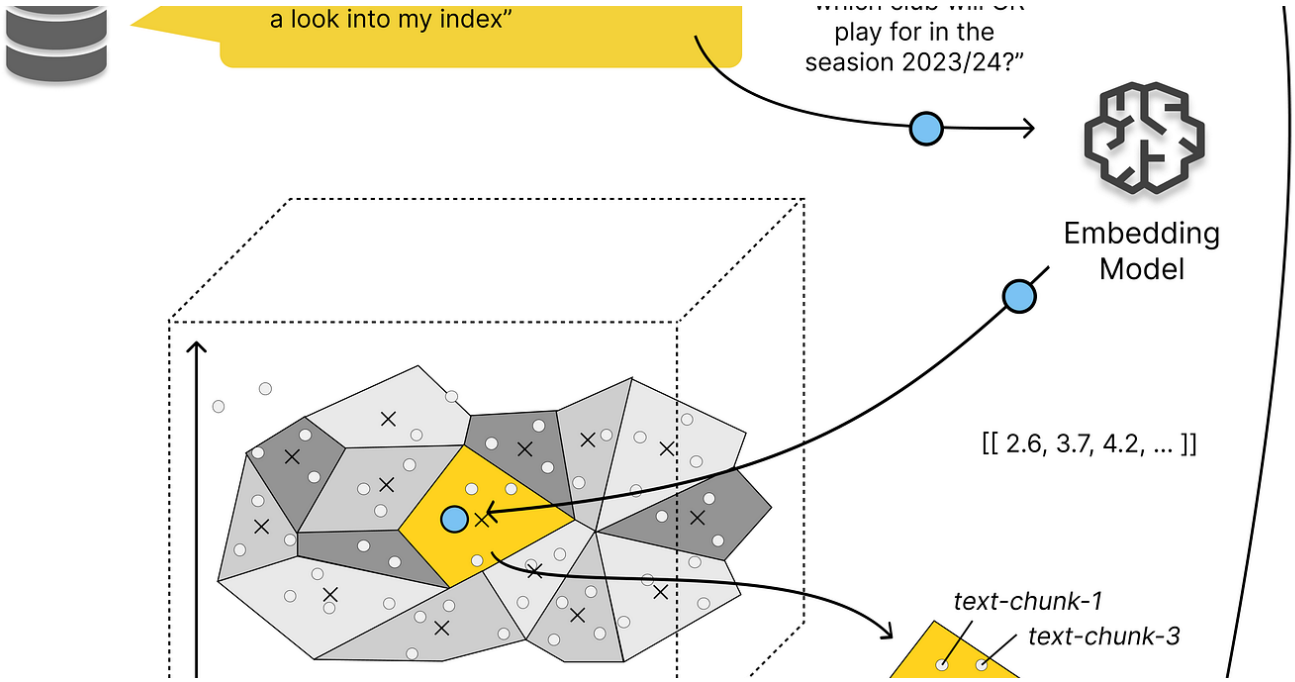
Rayyan Shaikh

Mastering BERT: A Comprehensive Guide from Beginner to Advanced in Natural Language Processing...

Introduction: A Guide to Unlocking BERT: From Beginner to Expert

19 min read · Aug 26

1.1K 14



Dominik Polzer in Towards Data Science

All You Need to Know about Vector Databases and How to Use Them to Augment Your LLM Apps

A Step-by-Step Guide to Discover and Harness the Power of Vector Databases

· 24 min read · 4 days ago

542 6

See more recommendations